E190AR: Lab 1 Report

Charles Dawson Peter Johnson

***Abstract*— This report describes the assembly, programming, and control of a robot test platform for E190AR. Section I describes the assembly of the robot platform. Section II discusses the programming and configuration of the Teensy microcontroller and XBee radio units. Section III discusses the structure of the `control.py` file used to control the robot, including comments on the function of each member function. Finally, Section IV describes the modifications made to `control.py` in order to implement support for the `angular.z` field of the \`cmd_vel` message. The complete `control.py` file with modifications and comments is attached as an appendix.**

I. Assembly

Throughout E190AR, we will use the Romi Pololu robot used in E160 (pictured in Fig. 1). By following the provided assembly instructions, we successfully:

1) Soldered header pins to the motherboard, two rotary encoder boards, Teensy 3.2 microcontroller, and XBee breakout board,
2) Attached the motherboard and Teensy/XBee breakout board to the robot chassis using small bolts (soldering the motherboard to the battery terminals in the chassis), then plugged the motors and rotary encoders into the motherboard,
3) Glued a mini breadboard to the chassis, then connected the Teensy/Xbee breakout board to the motherboard and three IR range finders according to the provided connection instructions, then bolted the IR range finders to aluminum standoffs attached to the chassis, and
4) Inserted 6 rechargeable AA batteries into the chassis.

The assembly process went smoothly, although we noticed difficulty when soldering to the ground pins on the motherboard, likely because the large ground plane in the motherboard acts as a heat sink and makes it more difficult to heat the solder pads and obtain a solid connection. This lead to the robot initially powering on and off when the power button was pushed. A quick check revealed that one of the solder connections on the motherboard was loose. Re-soldering the joint fixed this.

II. Programming & Configuration

To configure the two XBee radios used to communicate between the laptop running ROS and the robot platform, we designated one radio as the "Computer XBee" and the other as the "Robot XBee." We then used the XCTU software to configure each XBee with the parameters shown in Table 2. All other parameters were left at their default values. We labelled each XBee to ensure that the Robot and Computer radios were not mixed up; we attached the Robot XBee to the Teensy/XBee breakout board on the robot, while the Computer XBee was attached to the USB XBee dongle plugged into a laptop running ROS.

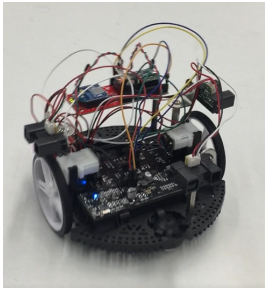

Fig. 1. Assembled Romi Pololu, the robotics platform used in E190AR.

XBee	Computer	Robot
Channel	C	C
PAN ID	13	13
Destination Address High	0	0
Destination Address Low	0	0
MY 16-Bit Source Address	0	C
Serial Interfacing	API enabled [1]	API disabled [0]
Coordinator Enable	Coordinator [1]	End Device [0]

Fig. 2. Configuration parameters for XBee radios

To configure the Teensy 3.2 microcontroller, we installed the provided `robot.ino` program on the Teensy using the Arduino IDE.

To configure the ROS program that will control the robot, we extracted the provided files into a `catkin` workspace. After installing the necessary Python packages for interfacing with the XBee radios, we ran the provided `control.py` ROS node program. Using the terminal output from this program, we verified that the `control.py` ROS node was able to communicate with the robot. Very little debugging was needed to ensure successful communication with the robot, although we did need to correct an error in the Robot XBee's PAN ID configuration parameter.

```
rostopic pub /cmd_vel geometry_msgs/Twist "linear:
    x: 0.4
    y: 0.0
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.0"
```

Fig. 3. Command used to instruct the robot to move forward.

Once we verified that the `control.py` ROS node was successfully communicating with the robot, we were able to execute the command shown in Fig. 3 from the Linux terminal. Based on discussions with our classmates, we determined that the command **must** be input with the correct line breaks and indentation; otherwise, the `control.py` node will not acknowledge the command. As noted in the instructions, this is easiest done by using tab to auto-complete and then changing the values.

III. Control Software Architecture

The robot is controlled via commands sent to its XBee radio by a ROS node defined in the `control.py` file. This file defines the class `botControl`, and when run, initiates a single instance of the `botControl` class. The `botControl` class defines a number of member functions. A commented version of the `control.py` file is shown in Appendix A.

IV. Control Software Modifications

The original `control.py` file did not support instructions to turn the robot, such as those shown in Fig. **??**. Initially, the `botControl` node only checked to see if the `CmdVel.angular.z` value is negative. If so, the robot was instructed to drive backwards; if not, the robot was instructed to drive forwards.

We modified the `control.py` file to change the `cmd_vel_callback` function to change the PWM signals sent to the motors based on the value in the `CmdVel.angular.z` field, as shown in Eqns 1 and 2. We also set the values indicating motor direction to be forward on the right wheel if $x + z > 0$ (reverse otherwise) and forward on the left wheel if $x - z > 0$ (reverse otherwise). As a result, the robot should be able to carry out compound maneuvers involving translation in x simultaneously with rotation in z. We tested our code by sending the command in Fig. 4 to the robot, and the robot spun in the desired direction (and changing the `angular.z` value to a negative number reversed the direction of spin). We observed that once one command was running, sending a second command stopped the first without causing the second to execute; re-sending the second command caused it to execute as expected.

$$PWM_R = |x + z|/255 \quad (1)$$

$$PWM_L = |x - z|/255 \quad (2)$$


```
rostopic pub /cmd_vel geometry_msgs/Twist "linear:
    x: 0.4
    y: 0.0
    z: 0.0
angular:
    x: 0.0
    y: 0.0
    z: 0.0"
```


Fig. 4. Command used to instruct the robot to move forward.

Appendix A: Commented & Modified Code

Note that line breaks denotes with a ↪ were inserted to make this document more readable and are not present in the original file.

```
#!/usr/bin/env python
import rospy
import rospkg
from xbee import XBee
import serial
import tf

from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion, quaternion_from_euler

rospack = rospkg.RosPack()

class botControl:
    ######################
    # constructor
    # creates an instance of the botControl object
    ######################
    def __init__(self):
        # create vars for hardware vs simulation
        self.robot_mode = "HARDWARE_MODE"#"SIMULATION_MODE"
        #self.control_mode = "MANUAL_CONTROL_MODE"

        # setup xbee communication, change ttyUSB0 to the USB port dongle is in
        if (self.robot_mode == "HARDWARE_MODE"):
            self.serial_port = serial.Serial('/dev/ttyUSB0', 9600)
            print(" Setting up serial port")
            try:
                self.xbee = XBee(self.serial_port)
            except:
                print("Couldn't find the serial port")

        print("Xbee setup successful")
        self.address = '\x00\x0C'#you may use this to communicate with multiple bots

        #init an odometry instance, and configure odometry info
        self.odom_init()

        #init log file, "False" indicate no log will be made, log will be in e190_bot/data folder
        self.log_init(data_logging=False,file_name="log.txt")

        rospy.init_node('botControl', anonymous=True)
        rospy.Subscriber("/cmd_vel", Twist, self.cmd_vel_callback)

        self.pubOdom = rospy.Publisher('/odom', Odometry, queue_size=10)

        #self.pubDistL = rospy.Publisher('/distL', ir_sensor, queue_size=10)
        #self.pubDistC = rospy.Publisher('/distC', ir_sensor, queue_size=10)
        #self.pubDistR = rospy.Publisher('/distR', ir_sensor, queue_size=10)
        self.time = rospy.Time.now()
        self.count = 0;

        self.rate = rospy.Rate(2)
```

```
    while not rospy.is_shutdown():
        self.odom_pub();
        self.rate.sleep();

# def ir_init(self):
# self.ir_L = ir_sensor()
# self.ir_C = ir_sensor()
# self.ir_R = ir_sensor()

############################
# odom_init
# Initialize odometry node using tf package broadcaster system
############################
def odom_init(self):
    self.Odom = Odometry()
    self.Odom.header.frame_id = "odom_wheel"
    self.Odom.child_frame_id = "base_link"
    self.odom_broadcaster = tf.TransformBroadcaster()
    self.encoder_resolution = 1.0/1440.0
    self.wheel_radius = .1 #unit in m, need re-measurement

############################
# log_init
# Initialize logging node
############################
def log_init(self,data_logging=False,file_name="log.txt"):
    self.data_logging=data_logging
    if(data_logging):
        self.file_name = file_name
        self.make_headers();

############################
# cmd_vel_callback
# Receive a CmdVel object from human input via the xBee
# CmdVel objects have a linear component and an angular component
# Translatetion along the x,y or z axis (x is forward, y to robot left, z vertical)
# Rotation along the x,y, or z axis (ccw is positive, cw is negative)
# Rotating right wheel forward produces ccw, left forward is cw
# creates a motor driver command for the arduino code to take in
############################
def cmd_vel_callback(self,CmdVel):
    if(self.robot_mode == "HARDWARE_MODE"):

        RCMD = CmdVel.linear.x + CmdVel.angular.z
        LCMD = CmdVel.linear.x - CmdVel.angular.z

        LDIR = 1 if (LCMD >= 0) else 0 #formatting due to weird inheritance typing in python
        RDIR = 1 if (RCMD >= 0) else 0

        RPWM =abs(int(RCMD/1*255)) #pwm needs integer values from 1-255, not negative
        LPWM =abs(int(LCMD/1*255))

        command = '$M ' + str(LDIR) + ' ' + str(LPWM) + ' ' + str(RDIR) + ' ' + str(RPWM) + '
            ↪ @'
        #print("subscriber working")

        #command = '$M ' + str(LDIR) + ' ' + str(LPWM) + ' ' + str(RDIR) + ' ' + str(RPWM) +
```

```
            ↪ '@'
        self.xbee.tx(dest_addr = self.address, data = command)

##################################
# odom_pub
# utilizes tf poses and transforms to format odometry data
# publishes and logs the data
# needs to be edited to perform the correct transforms
##################################
def odom_pub(self):
    if(self.robot_mode == "HARDWARE_MODE"):
        self.count = self.count + 1
        print(self.count)
        command = '$S @'
        self.xbee.tx(dest_addr = self.address, data = command)
        try:
            update = self.xbee.wait_read_frame()
        except:
            pass

        data = update['rf_data'].decode().split(' ')[:-1]
        data = [int(x) for x in data]
        encoder_measurements = data[-2:] #encoder readings are here, 2d array

        #print ("update sensors measurements ",encoder_measurements, range_measurements)

        #Update here, the code is totally non-sense

        #how about velocity?
        time_diff = rospy.Time.now() - self.time
        #self.last_encoder_measurementL =
        #self.last_encoder_measurementR =
        #self.diffEncoderL =
        #self.diffEncoderR =

        self.Odom.pose.pose.position.x = encoder_measurements[0]/10000.0 #this won't work for
            ↪  sure
        self.Odom.pose.pose.position.y = encoder_measurements[1]/10000.0
        self.Odom.pose.pose.position.z = .0
        quat = quaternion_from_euler(.0, .0, .0)
        self.Odom.pose.pose.orientation.x = quat[0]
        self.Odom.pose.pose.orientation.y = quat[1]
        self.Odom.pose.pose.orientation.z = quat[2]
        self.Odom.pose.pose.orientation.w = quat[3]

        # #https://wiki.ros.org/tf/Tutorials/Writing%20a%20tf%20broadcaster%20%28Python%29
        self.odom_broadcaster.sendTransform(
            (self.Odom.pose.pose.position.x, self.Odom.pose.pose.position.y, .0),
            tf.transformations.quaternion_from_euler(.0, .0, 1.57),
            rospy.Time.now(),
            self.Odom.child_frame_id,
            self.Odom.header.frame_id,
        )

        self.pubOdom.publish(self.Odom) #we publish in /odom topic

        #about range sensors, update here
```

```
            range_measurements = data[:-2] #range readings are here, 3d array
            #self.pubRangeSensor(range_measurements)

        if(self.data_logging):
            self.log_data();

        self.time = rospy.Time.now()

    # def pubRangeSensor(self,ranges):

    # #May be you want to calibrate them now? Make a new function called "ir_cal"
    # self.ir_L.distance = ir_cal(ranges[0])
    # self.ir_C.distance = ir_cal(ranges[1])
    # self.ir_R.distance = ir_cal(ranges[2])

    # self.pubDistL.publish(self.ir_L)
    # self.pubDistC.publish(self.ir_C)
    # self.pubDistR.publish(self.ir_R)

    ############################
    # make_headers
    # Format logging file
    ############################
    def make_headers(self):
        f = open(rospack.get_path('e190_bot')+"/data/"+self.file_name, 'a+')
        f.write('{0} {1:^1} {2:^1} {3:^1} {4:^1} \n'.format('R1', 'R2', 'R3', 'RW', 'LW'))
        f.close()

    ############################
    # log_data
    # logs data when odometry is being published
    ############################
    def log_data(self):
        f = open(rospack.get_path('e190_bot')+"/data/"+self.file_name, 'a+')

        # edit this line to have data logging of the data you care about
        data = [str(x) for x in [1,2,3,self.Odom.pose.pose.position.x,self.Odom.pose.pose.
            ↪ position.y]]

        f.write(' '.join(data) + '\n')#maybe you don't want to log raw data??
        f.close()

#Will initialize a new botControl object
if __name__ == '__main__':
    try:
        bot = botControl()

    except rospy.ROSInterruptException:
        pass
```